OPTIMISING ASSOCIATION RULE ALGORITHMS USING ITEMSET ORDERING

Frans Coenen and Paul Leng
Department of Computer Science,
University of Liverpool, UK
{frans,phl}@csc.liv.ac.uk

Abstract

Association-rule mining is a well-known method of Knowledge Discovery in Databases, aimed at identifying observable relationships in datasets the records of which can be represented as sets of items. The problem of extracting all possible association rules, however, is computationally intractable, so methods of reducing its complexity are of continuing interest. We here describe some results obtained from a method we have developed, which reduces the task by means of an efficient restructuring of the data accompanied by a partial computation of the totals required. The method is sensitive to the ordering of items in the data, and our experiments show how this property can be used as an implementation heuristic. The results we show demonstrate the performance gain for our method in comparison with a benchmark alternative, and the further gain from using the ordering heuristic.

KEYWORDS: KDD, Data Mining, Association Rules, Partial Support

1 Introduction

Knowledge Discovery in Databases (KDD) is concerned with the extraction of hitherto unrecognised and "interesting" information from within (usually large) datasets. Generally, this information takes the form of some relationship, or association, which can be observed between other categories of information in the data. These relationships can often be expressed in the form of *association rules* [2] which relate database attributes. Methods for identifying such rules have become an important area of research in KDD.

Association-rule mining has been developed, especially, to deal with problems such as supermarket shopping-basket analysis. The key characteristic here is that each database record (e.g. a shopping transaction) takes the form of a list of *items* occurring in the record (a set of items purchased). Many other kinds of data, such as census returns, also have a similar nature or can easily be converted into an equivalent form. The task then is to discover which sets of items typically occur together in records. From this we can infer relationships of the form $A \rightarrow B$, where A and B are disjoint sets of items. We interpret an association rule of this kind to mean "when the set of items A occurs in a record, it is likely that B will occur also".

To determine which of all such possible rules are of interest, a number of possible measures can be applied, of which the two most significant are the *support* and the *confidence*. The support for a rule describes the number of instances of the association in the dataset: that is, the number of records that include the set $A \cup B$. Almost all methods for determining association rules begin by excluding from consideration those sets the support for which is below some threshold value, expressed as a proportion of the total number of records. The confidence in a rule, then, is the ratio of the support for the association to the support for its antecedent: i.e. the confidence in the rule $A \rightarrow B$ expresses what proportion of records that include the set A also include B. Again, a rule is usually considered "interesting" only if its confidence exceeds some suitably-chosen threshold value.

Because the confidence in any rule can be determined immediately once the relevant support values have been calculated, association-rule mining invariably begins by seeking to establish the support for all those sets which may be involved in interesting rules, These so-called "frequent" sets are those which exceed the prescribed support threshold. The problem here is that the scale of this task is inherently exponential. If there are n possible items that may be included in a record, then an exhaustive computation requires the counting of the support for all 2^n possible sets of items. For most of the applications in which we are interested, n is likely to be 500 or more, making this computationally infeasible. All methods for determining association rules essentially focus on ways of addressing this problem.

We have developed a method for identifying frequent sets and efficiently computing their support totals, by starting with a pass of the database which performs a *partial* summation of the relevant totals, storing these in a set-enumeration tree structure (the *P-tree*). From this the final relevant support totals are computed, using a second tree structure (the *T-tree*) to store these. In this paper we present some experimental results that demonstrate the performance advantage gained from our approach. In particular, we focus on a characteristic of the method which makes its performance sensitive to the order in which the items in a set are considered. We describe the results of experiments with various data sets that show how this characteristic can be used to improve performance.

2 Background

Almost all algorithms for identifying frequent sets attempt to reduce the search space by computing support-counts only for those sets of items which are identified as potentially interesting. The best-known algorithm, "Apriori" [3], does this by repeated passes of the database, successively computing support-counts for single items, pairs, triples, and so on. Since any set of items can be "frequent" only if all its subsets also reach the required support threshold, the *candidate set* of sets of items is pruned on each pass to eliminate those that do not satisfy this requirement.

Apriori and related methods work well for cases for which the candidate sets being considered are relatively small and the algorithm terminates after a small number of database passes. For data in which individual records are densely-populated with items, however, candidate sets may become very large: if all the n single items reach the required threshold of support, for example, then all the $n(n-1)/2$ pairs will be candidates in the second pass, and so on. Large candidate-set sizes create a problem both in their storage requirement and in the computation required as each database record is examined. The implementation described for the Apriori algorithm stores the candidate set in a hash-tree, storing each candidate (a set of items) in a leaf node reached by hashing successively on each item. Then, as each database record is examined, it is necessary to increment the support for each candidate that is a subset of the record. These are located in the hash-tree by hashing successively on each item in the record, and examining all candidates thus found. Clearly, the time for this procedure will depend both on the number of items in a record and the size of the hash-tree.

Many subsequent methods have been developed to address the performance limitations of Apriori. These include strategies which begin by examining subsets of the database [10], or by sampling the database to estimate the likely candidate set [11]. These have the advantage that, in most cases, only one full pass of the database is needed, but the drawback that the candidate set derived is necessarily a superset of the actual set of frequent sets, so again may be very large. Other methods [13] [4] [5] aim to identify *maximal* frequent sets without first examining all their subsets. These algorithms cope better with densely-populated datasets than the other algorithms described, but again require multiple database passes. When the database is contained wholly in main memory, methods of this kind become more attractive [1]. However, it is fair to say that no known method provides a solution which scales satisfactorily to deal with very large and densely-populated databases, so this problem remains of active interest.

The fundamental performance problem of Apriori and related methods is the cost of dealing with all the subsets of densely-populated records, and of locating relevant candidates when the candidate-set becomes very large. In the next section we describe a method we have developed which addresses this problem by performing an initial, very efficient, *partial* computation of support totals, while at the same time reorganising the data to facilitate completion of the calculations needed.

3 Computing support via Partial totals

In the databases we are considering, each record takes the form of a set of individually different items, i.e. a subset of the total set of possible items. The restructuring we perform stores all *distinct* database records in the form of a set-enumeration tree [9]. A complete tree, for a set of items $\{A, B, C, D\}$, is illustrated in Figure 1. What is actually constructed, however, is a partial tree, including only those nodes that represent records actually occurring in the database, plus additional nodes that are created when two or more nodes share a leading subset. The scale of tree thus constructed is linear to the size of the database rather than exponential to the number of items.

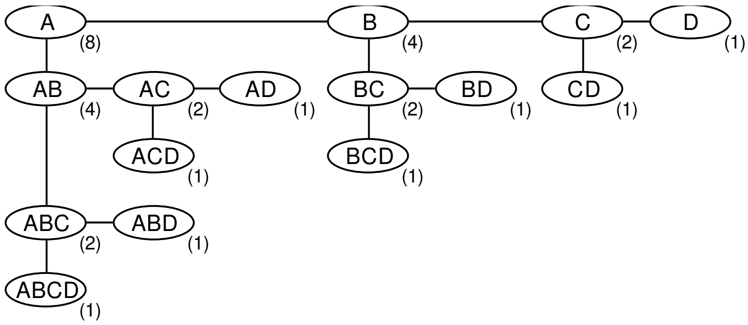


Figure 1: *Tree storage of subsets of* $\{A, B, C, D\}$

A detailed description of the algorithm for building this structure, which we call the *P-tree*, in a single database pass is given in [7]. One significant point to note, with respect to this paper, is that the itemset labels stored at each node (other than in the case of the top level nodes) exclude that part of the label represented by the node's parent. This reduces the overall storage requirements for the P-tree and consequently also contributes to gains in processing efficiency (this is discussed further in Section 5). The structure of the tree makes it easy and efficient to carry out, during its construction, a partial support-count of sets occurring within database records. It is easy to see from Figure 1 that the traversal of the tree necessary to locate any particular node will pass through, en route, all those subsets of the node-set that precede it in the tree ordering. Our tree-construction algorithm takes advantage of this to accumulate (incomplete) *interim* support-counts for all sets placed in the tree, incorporating the counts for all the successor-supersets of the set that are encountered. Figure 1 shows the counts that would arise from a database including exactly one instance of each of the 16 possible sets of items.

The advantage gained by this restructuring is that, by an efficient process in a single database pass, we have carried out a significant portion of the computation that would otherwise be performed repeatedly in each subsequent database pass. To illustrate this, consider the node BC in Figure 1, which has an interim support-count of 2, derived from one instance of BC and one of its successor-superset BCD. To complete the calculation of support for BC, we require to add also the counts for its predecessor-supersets, ABC and $ABCD$. But the contribution from $ABCD$ has already been included in the interim total for ABC, so the latter is the only addition needed to complete the summation of support for the set BC.

The set-enumeration tree structure has been used in a number of other methods, for storing candidates [4] [5], or frequent sets identified [1] rather than as a database surrogate. A rather similar concept to our *P-tree* has been described independently by [8]. Their *FP-tree*, has a more complex structure but quite similar properties to the P-tree, and is built in two database passes. The additional structure facilitates the implementation of an effective algorithm, "FP-growth", which successively generates subtrees from the FP-tree corresponding to each frequent item, to represent all sets in which the item is associated with its predecessors in the tree ordering. Recursive application of the algorithm generates all frequent sets. The drawback of this method is that the structural linkages of the FP-tree cause problems when dealing with data that cannot be wholly contained in main memory. The much simpler P-tree structure, conversely, is easily adaptable to non-memory- resident forms. Unlike the FP-tree method, which is the basis of a specific algorithm, the P-tree is essentially generic in that it is possible, in principle, to apply variants of almost any existing algorithm, using the P-tree as a replacement for the original database.

In order to illustrate and explore the performance gain from this, we have implemented a method based closely on the original Apriori algorithm, applying the Apriori procedure to the nodes in the P-tree rather than to the records in the original database. The principal advantage gained is that for any node of the P-tree, fewer subsets need be considered than would have been the case for the equivalent database record in the original Apriori. This is because support associated with predecessor-subsets of the node will already have been counted when considering the parent of the node in the tree. As an example, consider a record comprising the set of items $ABCD$. In the second pass of Apriori, its subsets to be considered are the pairs AB, AC, AD, BC, BD, and CD. However, within the P-tree, the count associated with $ABCD$ has already been included in the interim total for its parent ABC, and this will be added into the support totals for the subsets of ABC when the latter node is processed. Thus, when processing the node $ABCD$, we need only consider those of its subsets not covered by its parent, i.e. AD, BD and CD (AB, AC and BC having been considered when the node ABC was processed).

The structure of the P-tree also helps us to efficiently locate candidates whose support must be updated. For this purpose, we define a second structure, the *T-tree*, illustrated in Figure 2, to store the candidate set being considered. Again, the figure illustrates a complete tree, whereas in practice the actual tree constructed will include only those nodes that remain under consideration.

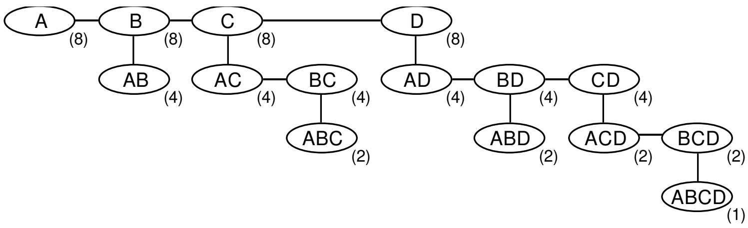


Figure 2: *Tree with predecessor-subtrees*

Each subtree of the T-tree stores predecessor-supersets of its root node. The significance of this is that it localises those sets which need to be considered when we examine the subsets of a P-tree node. Using the example above, when processing the P-tree node $ABCD$, we need to update the support total for AD, BD and CD, all of which are predecessor-supersets of the attribute D which is not included in the parent node. These sets are all located in the T-tree subtree rooted at node D. This allows us to define an efficient algorithm for counting the support of candidates stored in the T- tree. In this algorithm, which we call Apriori-TFP ("Total From Partial"), each node in the P-tree is examined, and its difference from its parent node is used to produce a "search set" S with an associated partial support total. The search set defines the branches of the T-tree which must be traversed. In the Apriori-TFP implementation, each pass of the algorithm builds a new level of the T-tree and uses this procedure to accumulate the support totals for its leaf nodes. At the end of the pass, those nodes that fail to reach the support threshold are removed, and the next level is built, adding only those nodes all of whose subsets remain in the tree. The process reproduces the Apriori methodology, terminating with a T-tree containing just the frequent sets. Note that the method requires each P-tree node to be visited just once on each pass, in any order, so dealing with non-memory-resident data poses no problems.

4 Itemset Ordering

The method we have described for computing support counts uses as a starting point an initial, incomplete computation stored as a set-enumeration tree. Although the actual algorithm described here to compute the final totals is based on the Apriori algorithm, the method itself is generic, in that, once the P-tree has been created, a variety of methods may be applied to complete the summation. Many of these methods, like the one we have illustrated, will be able to take advantage of the partial computation already carried out in the initial database pass to reduce the cost of further multiple passes.

Note, however, that the advantage gained from this partial computation is not equally distributed throughout the set of candidates. For candidates early in the lexicographic order, most of the support calculation is completed during the construction of the P-tree; for example, for the items of Figure 1, support for the sets A, AB, ABC and $ABCD$ will be counted totally in this first stage of the summation. This observation allows us to consider methods which maximise the benefit from this by a suitable ordering of the item set (a heuristic also used, in various ways, by [4], [6], [1] and [8]).

If we know, at least approximately, the expected frequency of each item, then an ordering which places more frequent items earlier in the set ordering would lead to a greater proportion of the summation being completed during construction of the P-tree (an efficient single database pass). The effect to be observed will be a P-tree which is more densely-populated at the left-hand side of the layout of Figure 1. This maximises the number of sets which have near-subsets as parents in the structure, hence reducing the number of subsets that need be considered.

To examine the effects of this, we have carried out a series of experiments. Our aim is, firstly, to investigate the performance gain achieved by using the P-tree and T-tree structures in place of the original database and Apriori hash-tree, i.e. to compare Apriori-TFP with the original Apriori. For this purpose, we have implemented a version of Apriori, as described in [3]. Both this, and our implementation of Apriori-TFP are straightforward, non-optimised Java programs. Secondly, we wish to investigate the effect on performance of ordering items in the way we have indicated.

For these experiments, we have used three different datasets. The first is a synthetic database constructed using the generator described in [3]. This uses parameters T, which defines the average number of attributes found in a record, and I, the average size of the maximal supported set. Higher values of T and I in relation to the number of attributes N correspond to a more densely-populated database. Here we show the results for the case where $T = 16$, $I = 8$, $N = 500$ and $D = 200,000$ (where D is the number of records).

This data, while important for comparison with published results of other methods, is relatively sparsely-populated and hence not a particularly demanding test of a method. Our second dataset is the "mushroom" available from the UCI Machine Learning Repository [12]. This is a densely populated set with $N = 129$ and $D = 7784$ (when normalised).

Finally, we have applied the methods to a "real" dataset; the "Fleet" database of an insurance company (the Royal Sun Alliance group), which records information on car use by staff. After converting to the required format, $N = 194$ and $D = 9000$. Note that this last data set, for reasons of confidentiality, is not publically available.

In each case two versions of the dataset were tested, one where the attributes were ordered according to frequency of single attributes (most frequent first), and one where the data set remained in its "raw" state. The characteristics of these data sets is summarised in Table 1.

	N	D	T	I	Size (Bytes)
T16.I8.N500.D200000	500	200,000	16	8	6,434,030
Mushroom.N129.D7784	129	7,784	-	-	358,064
Fleet.N194.D900	194	9,000	-	-	297,116

Table 1: Characteristics of test data sets

5 Results

5.1 P-tree Generation

Before examining the overall result of the association rule mining using Apriori-TFP it is appropriate to briefly consider the effect on P-tree generation. Table 2 shows the generation times and storage requirements for the three test data sets order and unordered. From the table the following can be observed:

- P-tree generation time is faster in all cases for the ordered against the unordered version of each of the data sets.
- For T16.I8.N500.D200000 and Fleet.N194.D900 the number of nodes in the P-tree increases because there are more nodes representing leading sub-strings then in the unordered case.
- In the case of Mushroom.N129.D7784 the number of nodes is reduced in the ordered case because this data set is a very dense set; therefore in the unordered form the P-tree contains many repetitions of certain sub-strings.
- In all cases the overall storage is reduced for the ordered versions, even where reordering results in more P-tree nodes, either because there were fewer nodes in the ordered version, or because the size of item-sets (i.e. the number of attributes represented by a particular item-set) stored at each node is significantly smaller. Remember that in the P-tree the attributes comprising an item set consist of the sub-string contained at the current node plus the substrings represented by its parent nodes.
- In addition, from comparison with Table 1 the overall P-tree storage for the ordered sets is less than that required for the unordered raw data, regardless of the "house keeping" overheads associated with the P-tree.

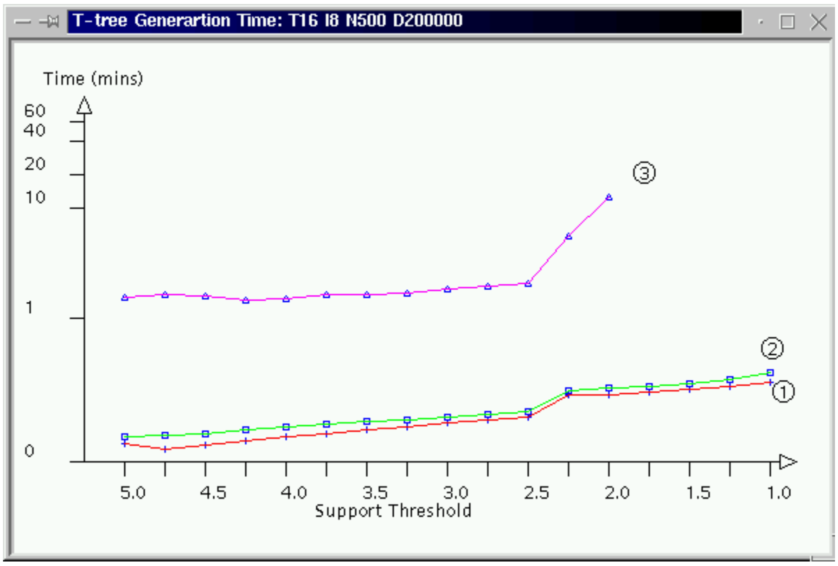
5.2 T16.I8.N500.D200000

Graphs 1 and 2 show sets of curves for time to generate the frequent sets, and the number of support-count updates involved given the T16.I8.N500.D200000 data set. Both graphs show three sets of curves: (1) Apriori-TFP using the

	T16.I8.N500. D200000		Mushroom.N129. 7784		Fleet.N194. 900	
	Unord.	Ord.	Unord.	Ord.	Unord.	Ord.
Gen. Time (mins)	9.79	8.70	0.39	0.36	0.34	0.31
Num. P-tree nodes	219,552	224,571	12,134	11,695	11,653	13744
Storage (Bytes)	6,583,844	6,374,156	204,304	181,704	304,256	265,612

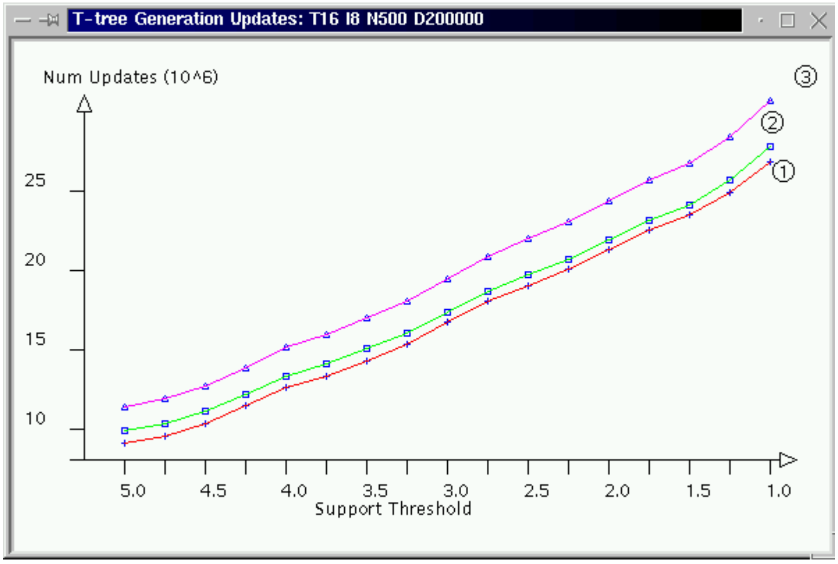
Table 2: P-tree generation characteristics

ordered version of the data set, (2) Apriori-TFP using the unordered version of the data set, and (3) frequent set generation using the traditional hash tree based Apriori technique. Graph 1 plots generation time in minutes against a range of support threshold values. From this graph it can be observed that Apriori-TFP, despite the overheads associated with the interaction with the P-tree data structure to produce the T-tree, outperforms the traditional hash tree approach. It can also be noted that reordering offers a slight further execution time advantage.



Graph 1 *Frequent set Generation time* `Apriori-TFP`: *(1) ordered P-tree to T-tree, (2) unordered Apriori-TFP, (3) unordered hash tree*

Graph 2 is arguably more informative in that it indicates the effectiveness of the P-tree technique and reordering in a manner independent of implementation optimisation and machine characteristics. Here, we are considering the number of support-count updates involved as the database or P-tree is scanned repeatedly. The number is lower when the P-tree is used, because of the reduced number of subsets that need be considered, as described in Section 3.

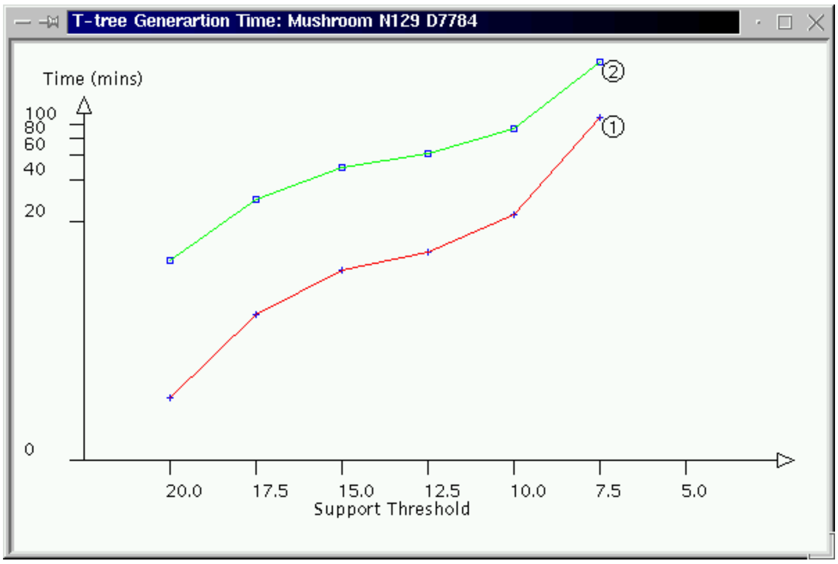


Graph 2 *Support count updates* `T16.I8.N500.D200000`: *(1) ordered P-tree to T-tree, (2) unordered Apriori-TFP, (3) unordered hash tree*

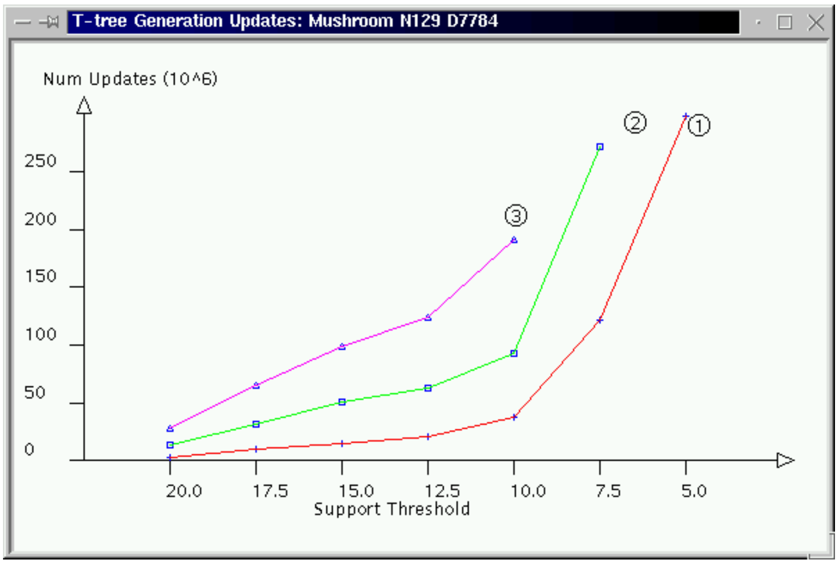
5.3 Mushroom.N129.D7784

Graphs 3 and 4 show similar sets of curves to those presented in the foregoing subsection but for the mushroom data set. The first graph presents data only for the P-tree to T-tree approach as the performance times for the hash tree exceeded the scale used for the graph. From this graph it can be seen that with respect to performance time reordering produces a significant advantage in this case (more so than when using T16.I8.N500.D200000). This is because the mushroom data set is densely populated, consequently the average length of an interesting set (I) is high. This also means that there are many levels in the T-tree each corresponding to an individual hash tree when using the traditional hash tree approach. (The effectiveness of the hash tree technique deteriorates as the magnitude of I increases.)

Graph 4 shows number of updates with respect to a range of support thresholds. Again, from this graph, it can be seen that reordering coupled with the P-tree to T-tree techniques offers significant advantages over the same technique used with unordered data. Note that in this case, unlike the experiments using the T16.I8.N500.D200000 data set, reordering results in a 50% gain. Similarly the Apriori-TFP algorithm significantly outperforms the hash tree approach regardless of ordering. Again the performance gains recorded when using Mushroom.N129.D7784 are a result of the dense nature of the dataset.



Graph 3 *Frequent set Generation time* `Mushroom.N129.D7784`: *(1) ordered P-tree to T-tree, (2) unordered Apriori-TFP*

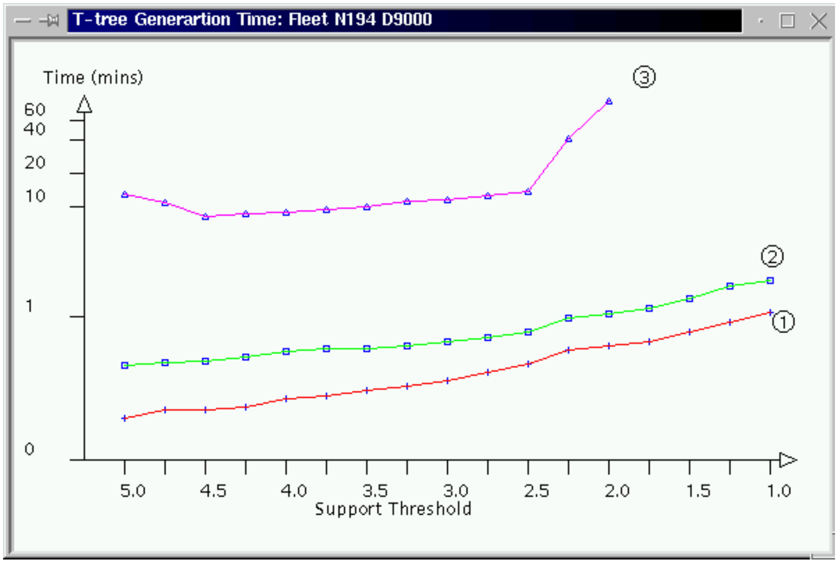


Graph 4 *Support count updates* `Mushroom.N129.D7784`: *(1) ordered P-tree to T-tree, (2) unordered Apriori-TFP, (3) unordered hash tree*

5.4 Fleet.N194.D900

Graphs 5 and 6 show similar sets to those presented previously for T16.I8.N500.D200000 and Mushroom.N129.D7784, but using a "live" data set provided by the Royal Sun Alliance Insurance group. Graph 5 shows that, with respect to execution time, the P-tree to T-tree technique offers significant advantages over the hash tree approach.

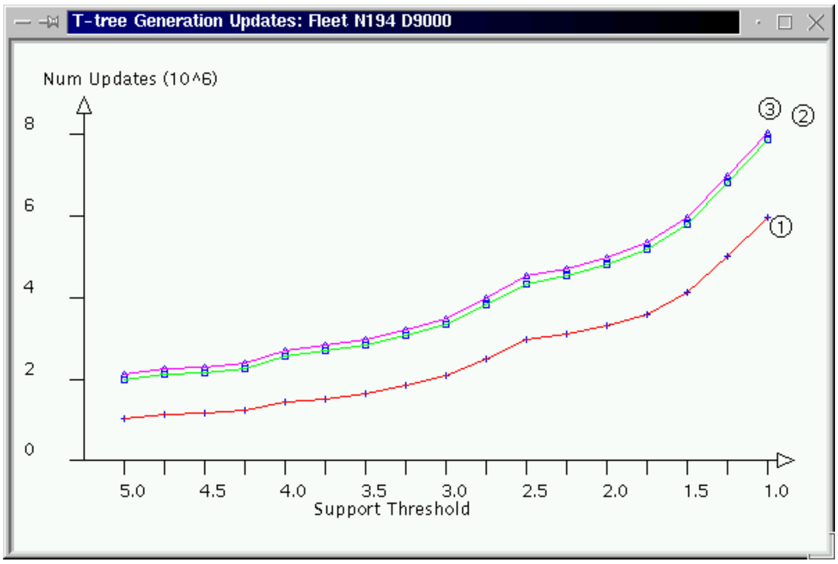
Interestingly, with respect to the number of updates using this data set, there was no marked improvement between the P-tree to T-tree approach against the hash tree approach until the data was reordered when a significant performance gain was realised.



Graph 5 *Frequent set Generation time* `Fleet.N194.D900`: *(1) ordered P-tree to T-tree, (2) unordered Apriori-TFP, (3) unordered hash tree*

6 Conclusions

We have in this paper described the results of a number of experiments we have carried out to investigate the performance of methods we have developed for extracting association rules from large databases. This problem is inherently complex computationally, and there is continued interest in finding methods which overcome the exponential scaling that follows when dealing with very densely-populated data. We have shown that our method consistently outperforms the "standard" benchmark algorithm, Apriori, even on relatively sparse data. On data that is more densely-populated, which poses the greatest challenges for all methods, the performance gain is significant. We have also shown that using a simple ordering heuristic gives rise to a further improvement in performance.



Graph 6 *Support count updates* `Fleet.N194.D900`: *(1) ordered P-tree to T-tree, (2) unordered Apriori-TFP, (3) unordered hash tree*

We observe, again, that our approach is generic in that the data structures we have described can be used as a basis for many other algorithms, and we expect that many of these will be susceptible to similar performance improvement. Investigating these algorithms is one avenue of research we are currently pursuing. A second is to further explore the use of the ordering heuristic. In particular we are investigating strategies which after ordering, partition the P-tree into regions (of different density of population) and apply different counting algorithms to each. We anticipate that hybrid strategies of this kind can give further performance gains, partly by reducing the size of partitions to a more efficiently-manageable level.

References

[1] Agarwal, R., Aggarwal, C. and Prasad, V. Depth First Generation of Long Patterns. Proc ACM KDD 2000 Conference, Boston, 108-118, 2000.

[2] Agrawal, R. Imielinski, T. Swami, A. Mining Association Rules Between Sets of Items in Large Databases. SIGMOD-93, 207-216. May 1993.

[3] Agrawal, R. and Srikant, R. Fast Algorithms for Mining Association Rules. Proc 20th VLDB Conference, Santiago, 487-499. 1994

[4] Bayardo, R.J. Efficiently Mining Long Patterns from Databases. Proc ACM-SIGMOD Int Conf on Management of Data, 85-93, 1998

[5] Bayardo, R.J., Agrawal, R. and Gunopolos, D. Constraint-based rule mining in large, dense databases. Proc 15th Int Conf on Data Engineering, 1999

[6] Brin, S., Motwani. R., Ullman, J.D. and Tsur, S. Dynamic itemset counting and implication rules for market basket data. Proc ACM SIGMOD Conference, 255-256, 1997

[7] Goulbourne, G., Coenen, F. and Leng, P. Algorithms for Computing Association Rules using a Partial-Support Tree. J. Knowledge-Based Systems 13 (2000), 141-149. (also Proc ES'99.)

[8] Han, J., Pei, J. and Yin, Y. Mining Frequent Patterns without Candidate Generation. Proc ACM SIGMOD 2000 Conference, 1-12, 2000.

[9] Rymon, R. Search Through Systematic Set Enumeration. Proc. 3rd Int'l Conf. on Principles of Knowledge Representation and Reasoning, 1992, 539-550.

[10] Savasere, A., Omiecinski, E. and Navathe, S. An efficient algorithm for mining association rules in large databases. Proc 21st VLDB Conference, Zurich, 432-444. 1995.

[11] Toivonen, H. Sampling large databases for association rules. Proc 22nd VLDB Conference, 134-145. Bombay, 1996.

[12] UCI Machine Learning Repository Content Summary. http://www.ics.uci.edu/ mlearn/MLSummary.html.

[13] Zaki, M.J., Parthasarathy, S. Ogihara, M. and Li, W. New Algorithms for fast discovery of association rules. Technical report 651, University of Rochester, Computer Science Department, New York. July 1997.